Containers: an Architecture for an Efficient Cluster Operating System

Renaud Lottiaux, Christine Morin, Geoffroy Vallée

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

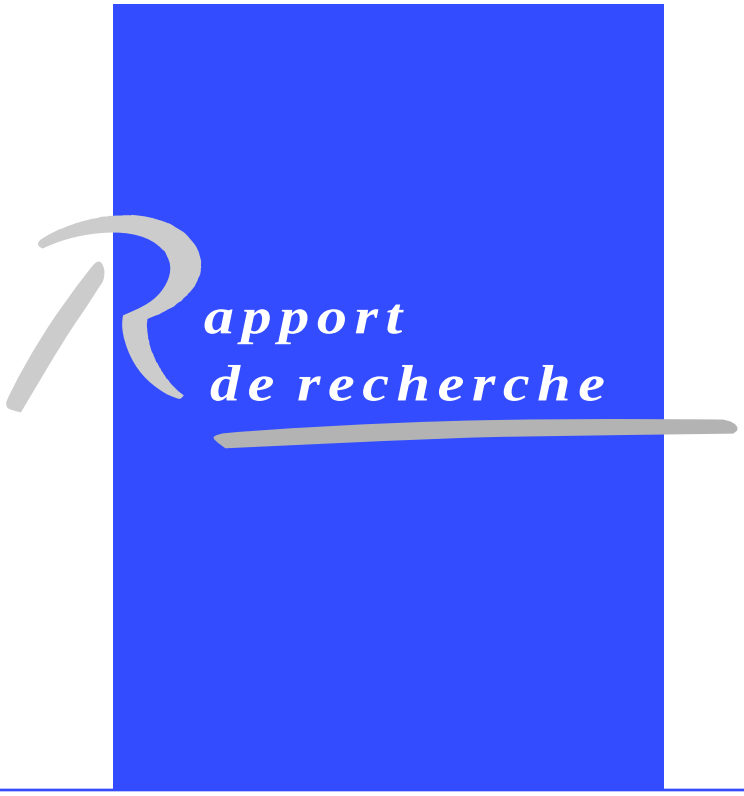
Containers: an Architecture for an Efficient Cluster Operating System

Renaud Lottiaux, Christine Morin, Geoffroy vallée

N°4384

Février 2002

THÈME 1



ISSN 0249-6399 ISRN INRIA/RR--4384--FR+ENG

Containers: an Architecture for an Efficient Cluster Operating System

Renaud Lottiaux*, Christine Morin†, Geoffroy vallée‡

Thème 1 — Réseaux et systèmes
Projet PARIS

Rapport de recherche n ° 4384 — Février 2002 — 28 pages

Abstract: The lack of a single system restricts the use of parallel processing on clusters. We propose an approach for building an efficient single system image cluster Operating System (OS). The proposed system is based on container, a concept which provides the abstraction of shared memory segment at the cluster scale. Small pieces of software, called linkers, are used to integrate containers in an existing single node OS. Hence, the existing OS programming API is kept unmodified but the underlying OS services can take benefit of all cluster resources. Our approach has been validated by a prototype based on Linux. Existing applications running on symmetric multiprocessors (SMP) on top of Linux can be executed on top of our cluster OS without modification.

Key-words: Cluster, operating system, distributed shared memory, global resource management

(Résumé : tsvp)

* INRIA {rlottiau,cmorin,gvallee}@irisa.fr
† Université de Rennes 1
‡ INRIA/EDF

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : 02 99 84 71 00 - International : +33 2 99 84 71 00
Télécopie : 02 99 84 71 71 - International : +33 2 99 84 71 71

Les conteneurs : une nouvelle approche pour la conception d'un système d'exploitation pour grappe

Résumé :

L'absence de système d'exploitation spécifique aux grappes limite leur utilisation dans le cadre du calcul parallèle. Nous proposons une approche pour la conception d'un système d'exploitation à image unique efficace pour grappe. Le système proposé est fondé sur le concept de conteneur, qui fournit l'abstraction de segments de mémoire partagée à l'échelle d'une grappe. De courts éléments logiciels appelés lieurs, sont utilisés pour intégrer les conteneurs dans un système d'exploitation existant. Ainsi, il est possible de conserver l'interface de programmation de ce système, tout en tirant partie de toutes les ressources de la grappe. Notre approche a été validée par un prototype fondé sur Linux. Grâce à ce système, il est possible d'exécuter sur une grappe sans aucune modification du code, des applications existantes conçues pour des multiprocesseurs symétriques (SMP).

Mots-clé : Grappe, système d'exploitation, mémoire partagée répartie, gestion globale des ressources

1 Introduction

Clusters are now more and more widely used as an alternative to parallel computers as their low price and high potential performance make them really attractive for the execution of data servers or simulation applications. However, clusters suffer of a lack of dedicated operating system hiding the distribution of resources. As a consequence, clusters are difficult to use and to program.

Single system image (SSI) is a solution to ease cluster programming and to reach functionalities offered by parallel computers. An SSI offers users and programmers the illusion that a cluster is a single high performance and highly available computer instead of a set of independent machines interconnected by a network. An SSI should offers four properties: (1) resource distribution transparency, i.e. offering the illusion that each kind of resource is unique and shared, (2) high performance, (3) high availability, i.e. tolerating node failures and allowing applications checkpoint and restart and (4) scalability, i.e. dynamic system reconfiguration, node addition and eviction transparently to applications.

In this paper we focus on the first property: resource distribution transparency. A large number of research projects aim at providing this property. However, most of the existing systems focus on global management of one kind of resource (processor [4], memory [11, 2, 6] or disk [3, 5]) without considering other kinds of resources.

A cluster can appear as a single machine to all applications only if each kind of resource appears as a single and shared resource. To achieve this goal, we could imagine to integrate several existing systems within the same cluster in order to globalize each resource. However, this naive solution is not realistic. The resulting system would consist of many redundant software mechanisms and would be a complex system. Such a complexity would make the system difficult to maintain and weakly reliable. Moreover, each sub-system taking local decisions would interfere with the decisions taken by the other sub-systems.

In this paper, we present an operating system level solution to factorize the software components used by existing solutions. By factorizing software components, we are able to easily design an operating system managing all resources in a cluster. However, the design of a new operating system is a long and complex task. As a consequence, we propose the modification of an existing operating system, rather than the design a new system from scratch. We call this existing system the *host system*.

Our solution is based on the concept of *container* and *linkers*. The container mechanism provides global management of the cluster physical memory and gathers the two main mechanisms used by the existing global resource management systems,

which are: (1) memory pages migration / duplication and (2) ensuring replicated pages coherence. Linkers are software pieces allowing to easily implement distributed operating system services, based on the use of containers. Containers and linkers allow the design of services as complex as a shared virtual memory (SVM) [11, 2], a cooperative cache [5] or a distributed file mapping system and how containers can ease process migration design [4]. Our approach have been validated with a prototype called GOBELINS based on LINUX kernel.

The remainder of this paper is organized as follows: section 2 is devoted to background. Containers and linkers are presented in section 3. Section 4 describes distributed services on containers. The prototype implementation is presented in section 5 and performance results are analyzed in section 6. Section 7 concludes.

2 Background

Since the late 80's, we assist to the emergence of systems combining the management of several resources. Sprite [12] is probably one of the first systems which has integrated within the same OS the management of several resources, namely disks and processors. Sprite is an operating system written from scratch, integrating a distributed file system ensuring a full transparency of file localization and a process migration mechanism. The advantage of combining these two mechanisms in an unique system is to offer a better transparency of the resource distribution and to limit the constraints met at the frontier between a resource managed globally and a resource managed locally. However, Sprite does not offer any global memory management mechanism, does not allow the migration of threads and has a weak distributed file cache strategy.

Millipede [7] is a user level system implemented on top of Windows NT. It combines global memory and processor management. Just like in Sprite, this approach allows a good transparency to the distribution of resources and limits the constraints met at the frontier between the management of processors and memories. Indeed, in the Millipede system it is possible to migrate threads, thanks to the use of an SVM. Nevertheless, the user level implementation of Millipede introduces many programming constraints and limits its performance. For instance, data shared through the SVM must be explicitly handled by the programmer thanks to a set of specific functions. Moreover, the use of system calls is strongly limited. Lastly, the Millipede system cannot modify the file cache, page replacement or disk management mechanisms of the underlying Windows NT system.

The Genesis system [8] is close to Millipede, in the sense that it aims at offering a single system image based on global memory and processor management. The originality of this system is to offer to the programmer the choice to use a shared memory and/or message passing programming model. This system has been implemented from scratch and offers more effective services and a better distribution transparency than Millipede.

Several others systems like Nomad [13], UnixWare NonStop Cluster [17] or SSIC [1] have been proposed. However, it appears that no system ensures a global management of every cluster resource, allowing to really offer the view of a single machine on top of a cluster. We present in the remainder of this paper the design of a generic service allowing to globally manage all the resources of a cluster and which can be easily integrated in an existing operating system.

3 Containers and Linkers

In a cluster, each node executes its own operating system kernel, which can be coarsely divided into two parts: (1) system services and (2) device managers. We propose a generic service inserted between the system services and the device managers layers called *container* [15]. Containers are integrated in the core kernel thanks to *linkers*, which are software pieces inserted between existing device managers and system services and containers. The remainder of this section describes container and linker mechanisms. The key idea is that container gives the illusion to system services that the cluster physical memory is shared as in an SMP machine.

3.1 Container Definition

A container is a software object allowing to store and share data cluster wide as a COMA [9] architecture does. Container is a kernel level mechanism completely transparent to user level software. Data are stored in a container on host operating system demand and can be shared and accessed by the host kernel of other cluster nodes. Pages handled by a container are stored in page frames and can be used by the host kernel as any other page frame. Container pages can be mapped in a process address space, be used as a file cache entry, etc.

By integrating this generic sharing mechanism within the host system, it is possible to give the illusion to the kernel that it relies on top of a physically shared memory. On top of this virtual physically shared memory, it is possible to extend to a cluster scale traditional services offered by a standard operating system (see fig-

ure 1). This allows to keep the OS interface known by users and to take advantage of the existing low level local resource management.

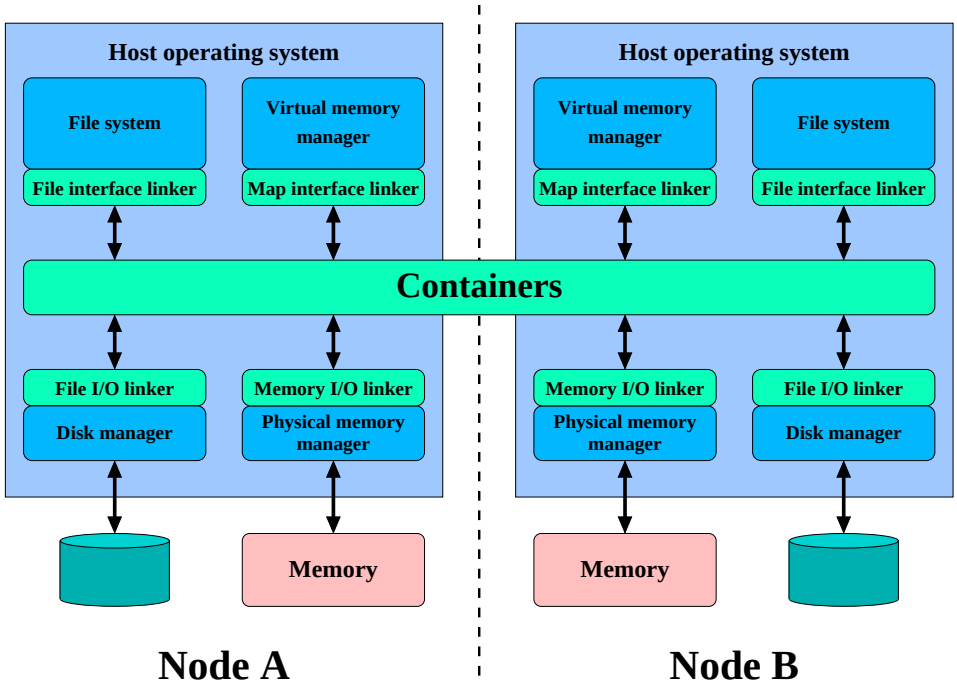


Figure 1: *Integration of containers and linkers within the host operating system*

The memory model offered by containers is sequential consistency implemented with a write invalidation protocol similar to [11]. This model is the one offered by physically shared memory. Moreover, an injection mechanism similar to [6] is used to balance memory usage and avoid (or delay) disk swapping.

3.1.1 Container Interface

The access to and management of container pages are carried out thanks to four interface functions. This interface is used by linkers in order to connect high level services to containers.

- *find_page*: checks if a given page of a container is present in local memory.

- *get_page*: places a copy of a given page of a container in local memory. If a copy of the requested page exists in a remote node memory, a copy of this page is placed in local memory. If there is no copy of the requested page cluster wide, a call to a linker is done to request the creation of the page.
- *grab_page*: places a *unique* copy of a given page of a container in local memory. This function behaves as the *get_page* except that all copies (except the local one) are invalidated thanks to linkers function.
- *flush_page*: removes a container page from the local memory following a page replacement. If a remote copy of the page exists, the local copy is simply discarded. If the local copy is the only one in the cluster, it is injected in the memory of a remote node. Finally, if the injection is not possible due to lack of memory, the page is swap out thanks to a linker function.

3.2 Linkers

Many mechanisms in a core kernel rely on the handling of physical pages. Linkers divert these mechanisms to ensure data sharing through containers. To each container is associated one or several high level linkers called *interface linkers* and a low level linker called *input/output linker*. The role of interface linkers is to divert device accesses of system services to containers while I/O linkers allow containers to access a device manager.

3.2.1 Connecting a Container to System Services

System services are connected to containers thanks to interface linkers. An interface linker changes the interface of a container to make it compatible with the high level system services interface. This interface must give the illusion to these services that they communicate with traditional device managers. Thus, it is possible to "trick" the kernel and to divert device accesses to containers. It is possible to connect several system services to the same container. For instance it is possible to map a container in the address space of a process P1 on a node A and to access it thanks to a read/write interface within a process P2 on a node B.

3.2.2 Data Input/Output in Containers

During the creation of a new container, an input/output linker is associated to it. The container then stops being a generic object and allows to share data coming from

the device it is linked with. The container is said to have been instanciated. For each semantically different data to share, a new container is created. For instance, a new container is used for each file to share and a new container for each memory segment to share or to be visible cluster wide.

Just after the creation of a container, this one is completely empty, i.e. it does not contain any page and no page frame contains data from this container. Page frames are allocated on demand during the first access to a page. Similarly, data can be removed from a container when this one is destroyed or in order to release page frames when the physical memory of the cluster is saturated. These actions are performed by input/output linkers on containers demand through the following interface:

- *first_touch*: this function allocates a page frame during the first access to a container page. If the linker is associated with a storage device, the data is loaded from this device within the allocated page frame.
- *flush_page*: this function frees a page frame on a page replacement algorithm demand.
- *free_page*: this function frees a page frame when a page is definitively removed from a container. This situation occurs when a container is destroyed.
- *invalidate_page*: this function frees a page frame from local memory when a page is invalidated by the coherence protocol.

4 Scaling Operating System Services to a Cluster Scale with Containers

To provide on a cluster system services similar to those present in a shared memory parallel computer, we extend to a cluster scale traditional system services. To extend services using containers, we need to define two kinds of interface and I/O linkers: memory linkers and file linkers. We detail the design of these linkers in the remainder of this section. We then details who we use linkers and containers to extend existing system services.

4.1 Design of Interface Linkers

We describe in this section the design of two types of interface linkers: a linker offering a *mapping* interface and a linker offering a *read/write* interface. The mapping

interface is mainly used by a kernel to map memory segments or files in a process address space. The read/write interface is the basic interface offered to users to access files.

4.1.1 Memory Map Linker

Host Kernel Architecture

A process memory image consists of a set of segments corresponding to the mapping of devices or physical memory in its address space. The text segment corresponds to the mapping of file, the data segment corresponds to a piece of physical memory mapped, etc.

In a kernel, the mapping interface generally consists of two functions: (1) *first_touch*, which allocates a page frame during the first access to a page and (2) *copy_on_write*, which creates a new page frame and copy data from the original page in the new one. The physical memory manager ensures the management of page frames through two functions: (1) *alloc_page* which allocates a page frame and (2) *release_page* which releases a page frame.

Design of the Memory Map Interface Linker

When a memory segment is linked to a container, the kernel *first_touch* and *copy_on_write* functions are replaced by the two memory map linker functions : *Memory_Linker_First_touch* and *Memory_Linker_Copy_on_Write*.

Figure 2 presents the algorithm of the *Memory_Linker_First_touch* function. During a write access, the linker calls the associated container *grap_page* function to get a writable copy of the page and return its physical address. During a read access, the linker checks if the page is present in local memory thanks to the *find_page* function. If the page is not present, the *get_page* function is called to get a copy in local memory. Finally, the address of the page is returned.

Figure 3 presents the algorithm of the *Memory_Linker_Copy_on_Write* function. When a memory segment is linked to a container, the *copy_on_write* function is called when a write access is performed on a read only page. In this case, the linker gets a writable copy of the page thanks to the *grap_page* function and returns the address of the page.

```
Memory_Linker_First_Touch :
{
    if write access on p then
      page := grab_page (p) ;
    else
      page := find_page (p) ;
      if page = NULL then
        page := get_page (p) ;
      end if
    end if
    return (page) ;
}
```

Figure 2: First_Touch algorithm

```
Memory_Linker_Copy_On_Write :
{
    page := grab_page (p) ;
    return (page) ;
}
```

Figure 3: Copy_on_Write algorithm

4.1.2 File Linker

Host Kernel Architecture

A file system is generally composed of two levels: a virtual file system (VFS) and a set of physical file systems. A unique file cache is used by the VFS to keep in memory the most frequently used pages coming from the different file systems. Each access to a file goes through the file cache thanks to the *lookup_page* kernel function. This function checks if the requested page is present in the file cache. If the cache cannot satisfy this request, the VFS asks the associated file system to read the page from disk thanks to the *read_page* kernel function.

During a write access to a file, written data is stored in the file cache before being written back to disk. In this case, the VFS checks if the page to be written is present in the cache thanks to the *lookup_page* function. If the page is not in the cache, a new entry is created in the cache. The data is then copied from the user buffer to the cache. Periodically, modified data in the cache is updated on disk thanks to the *write_page* function from the associated file system.

Design of the File Interface Linker

The read and write functions systematically pass through the *lookup_page* function. By diverting every access to this function, it is possible to divert every file access to containers. If interface linker functions ensure to always return a page frame, the *read_page* function will never be called by the VFS. Thus, it is not necessary for the interface linker to divert this function.

Finally, the *write_page* function used by the VFS to update storage units is diverted in order to carry out the backup of pages from cache on the node hosting the associated devices.

When a file is linked to a container, the kernel *lookup_page* and *write_page* functions are replaced by the two file linker functions: *File_Linker_Lookup_Page* and *File_Linker_Write_Page*.

Figure 4 presents the algorithm of the *File_Linker_Lookup_Page* function. When a file is linked to a container, a call to the *lookup_page* function places in local memory data coming from the container associated to the file. The *lookup_page* function determines which container the file is associated with, and carries out one the following actions according to the type of access.

```
File_Linker_Lookup_Page :
{
    if write access on p then
      page := grab_page (p) ;
    else
      page := find_page (p) ;
      if page = NULL then
        page := get_page (p) ;
      end if
    end if
    return (page) ;
}
```

Figure 4: Lookup in a file cache algorithm

```
File_Linker_Write_page :
{
    if node hosting file linked to p = local node then
      write_page (p) ;
    end if
}
```

Figure 5: Storage device updating algorithm

- In the case of a write access, the linker calls the *grap_page* function in order to get a writable copy of the page and returns its address.
- In the case of a read access, the linker checks if the page is present in local memory thanks to the *find_page* function. If the page is present, its address is returned. Otherwise, the *get_page* function is called.

Figure 5 presents the algorithm of the *File_Linker_Write_Page* function. Periodically, the VFS scans all the pages present in the file cache and writes back to disk dirty pages. Thus, the role of the interface linker is simply to relay *File_Linker_Write_Page* function calls to the *write_page* function of the associated file system when this makes sense, i.e. on the node hosting the device on which the file is stored.

4.2 Design of Input/Output Linkers

We detail in this section the design of two kinds of input/output linkers: one allowing the access to the memory resource and the other one ensuring access to block devices.

4.2.1 Memory I/O Linker

The memory I/O linker ensures input and output of physical memory pages in and out of containers. When a container is linked to a memory I/O linker, it becomes a *memory container*. The memory I/O linker is very simple since it consists in allocating and releasing page frames like the host kernel does for the management of memory segments.

- *first_touch*: allocates a page frame.
- *free_page*: releases a page frame.
- *flush_page*: calls the host kernel replacement function and releases the page frame.
- *invalidate_page*: releases a page frame.

4.2.2 File I/O Linker

The file I/O linker ensures input and output of file data in and out of containers. When a container is linked to a file I/O linker, it becomes a *file container*. The file I/O linker actually consists in carrying out accesses to the storage unit hosting files associated with containers.

- *first_touch*: this function allocates a new entry in the file cache and loads the page from the storage unit. The address of the allocated page frame is then returned.
- *free_page*: this function updates the disk copy if necessary, removes the entry in the file cache and releases the associated page frame.
- *flush_page*: the replacement of a page from the file cache does not require to write it in the swap area on disk. The page can indeed be simply written back on disk on the node hosting the corresponding file. If the disk is up to date, no disk access is needed. Thus, the *flush_page* function performs the same operation as the *free_page* function. The page frame is simply released after having updated the associated device if necessary.

- *invalidate_page*: the invalidation of a page from a file container makes the page locally unreachable. The page is removed from local memory, but there exists at least another copy in the cluster. Then, it is not necessary to update the associated device. The *invalidate_page* function removes the corresponding entry in the file cache and releases the corresponding page frame.

4.3 Design of Distributed System Services

We now present how containers and linkers are used to provide at a cluster scale the same high level services as those provided by a single node OS.

4.3.1 Shared Virtual Memory

The virtual memory sharing service of an OS allows to share data between threads or between processes through a system V segment for instance. A shared virtual memory extends this service to a cluster scale.

A shared virtual memory allows several processes running on different nodes to share data through their address space. Providing this service requires to ensure three properties: (1) data sharing between nodes, (2) coherence of replicated data and (3) simple access to shared data thanks to processor read/write operations.

The container service ensures the two first properties. The third one is ensured by the mapping interface linker. Thus, mapping a memory container in the virtual address space of several processes via a mapping linker leads to a shared virtual memory.

When a process page fault occurs, the memory map interface linker diverts the fault to containers. The container mechanism places a copy of the page in local memory and ensures the coherence of data. Lastly, the map interface linker maps the local copy in the address space of the faulting process and changes virtual page access right according to the rights of the page in the container.

4.3.2 File Mapping

The file mapping service of an OS allows to map a file in the address space of one or several processes, or in the address space shared by a group of threads. Extending this service to a cluster scale suppose to allow to map a file in a process address space whatever its execution node or in the shared memory of a group of threads. This is done by mapping a file container in the virtual address space of a process thanks to a mapping interface linker. Moreover, it is possible to map a file container in the

address space of one or more processes running locally or on a remote node and in shared virtual memory.

4.3.3 Cooperative File Cache

The file cache service of an OS allows to keep in memory most frequently used data coming from files loaded by every processes running or having run on the machine. The extension of this service to a cluster scale is a cooperative file cache.

No specific mechanism is needed to design a cooperative file cache in our system. Indeed, when a file is stored in a container, this one behaves naturally like a cooperative cache [5]:

- A container not being related to a specific process, it can survive to the death of a process and continue to store data loaded by this process. This data can then be used by other processes without requiring any disk access.
- Data loaded on a node can be used on another node through a container. Data referenced by a process is copied automatically in the file cache of its execution node by the container mechanism.
- Coherence of data replicated in different file caches is automatically ensured by the container coherence mechanism.
- The amount of memory used by the file cache adapts automatically according to processes memory usage. If a process has significant requirements in term of memory, file cache data is automatically removed to free memory for processes data. However, file cache data is not lost if there exists other copies on a remote node or if they can be injected in the memory of another node.

4.4 Container based Process Migration

The process scheduling service of an OS allows to schedule processes on available processors of the architecture. The extension of this service to a cluster scale is a global scheduler. To place or move processes on available processors of the cluster, a process migration mechanism is needed. The migration of a process from a node to another combines two kinds of mechanisms: (1) a process transfer mechanism and (2) post-transfer execution mechanisms.

The process transfer mechanism consists in extracting the process context (registers, stack, kernel data structure, etc) from the kernel, transferring this context to

the target node, inserting the context in the target kernel and rebuilding connections between the context and kernel data structures and finally restarting the process. This phase need specialized mechanisms for which containers cannot be used.

The post-transfer execution mechanisms ensure to the migrated process a transparent access to resources used on the initial node. Two kinds of devices can be identified: char devices (keyboard, printer, etc) and block devices (memory, disk, etc). Access to remote char devices is ensured by specific mechanisms but access to remote blocks devices can be ensured automatically by containers. Thanks to containers, problems as difficult as the migration of the address space and the link with open files are solved easily without designing any new specific mechanism.

4.4.1 Migrating the Address Space

Let's take the example of a process with an address space composed of the following segments:

- **Text segment**. Process code file mapped in the process address space. This segment is read only.
- **Initialized data segment**. Process code file mapped in the process address space with an offset placing the beginning of mapping on the file area containing initialized data. Write operations to this segment are not reflected on the associated file.
- **Non initialized data segment**. Piece of physical memory mapped in the process address space.
- **Stack segment**. This segment is a piece of physical memory mapped in the virtual address space of the process.

During the process transfer phase, a new container is created for each memory segment before the extraction of the process context (see figure 6). As every file is placed in a container when it is open, it is not necessary to create new containers for file mapped segments.

Once containers have been created, the process context is extracted and transfered to the target node. No data contained in the memory segments is migrated. During the post-transfer execution phase, an access to an invalid page in the process address space raises a page fault diverted to the corresponding container by the memory linker. The corresponding page is then copied by containers from the process source node to the current process execution node.

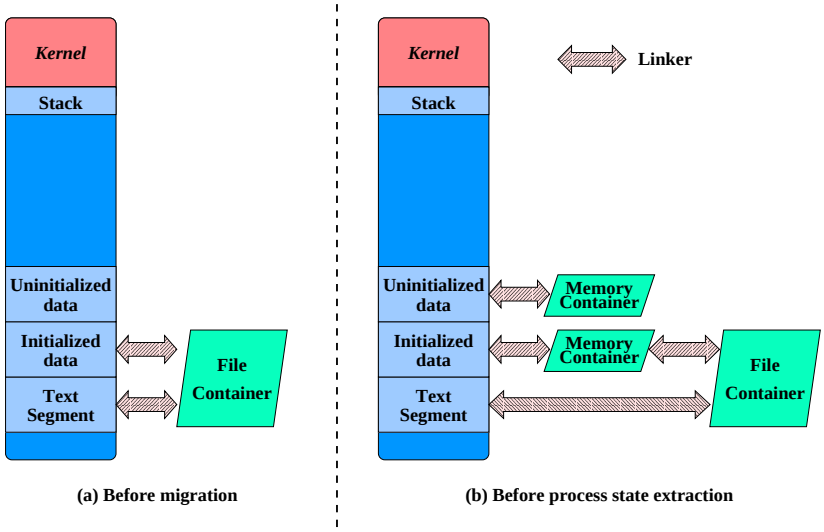


Figure 6: *Containers linked to process segments*

4.4.2 Access to Open Files After Migration

Let's say that our example process opened a local file before migration through the standard file interface.

During the process transfer phase, the list of open file descriptors and their corresponding container identifiers are transfered to the target node. On the target node, the connection between each open file descriptor and the associated container is restored.

During the post-transfer execution phase, accesses to an open file are diverted to containers by the file interface linker. Through containers, data located in the file cache of the initial node is automatically copied in the local cache of the target machine. Moreover, data of files opened by the process may already be present in the target machine cache due to the natural cooperative cache behavior of containers.

In Mosix for instance, the local cache is never used for a migrated process. As a consequence, each file access needs to transfer data from the node hosting the file. With containers, when file data has been placed in local file cache it can be used like traditional file cache data for future file accesses without any new network transfer, increasing file access performance.

5 Implementation

A prototype named GOBELINS has been implemented by integrating container and linker mechanisms within the LINUX operating system. In this system, a shared virtual memory, a cooperative file cache, a distributed file mapping system and a process migration mechanism have been implemented on the basis of containers and linkers.

The implementation of containers represents 1700 lines of code whereas the implementation of linkers, and thus of the above system services, represents only 1100 lines of code. The modification of the LINUX core kernel remains extremely limited, since less than 200 lines of code were modified or added.

The implementation of GOBELINS relies on three kernel modules: a communication module called GIMLI, a container manager module called GANDALF and a migration module called ARAGORN (see figure 7). Description of process migration mechanisms implemented in GOBELINS can found in [16].

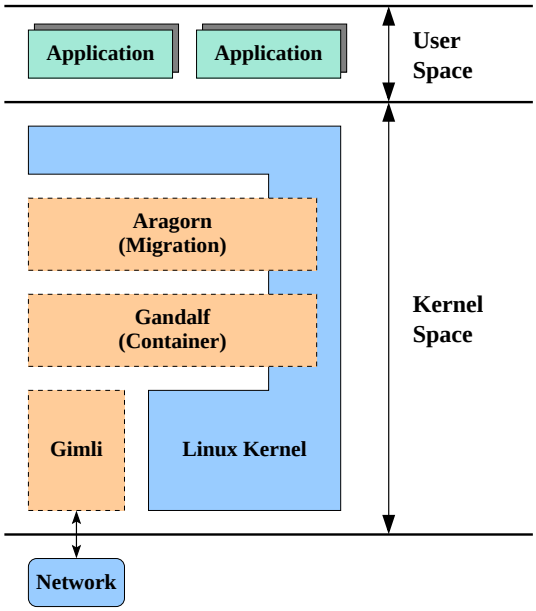


Figure 7: *Architecture of the* GOBELINS *operating system*

5.1 Gimli Implementation

The implementation of containers and other GOBELINS services requires to exchange data between kernels running on cluster nodes. Linux offers two types of communication mechanisms within the core kernel: sockets and RPC. However, they offer poor performance and a complex programming interface. Thus, we have implemented a portable communication library allowing reliable and high performance kernel to kernel communications [14]. This library offers a unique and simple interface (send/receive, active messages and pack/unpack) on top of Ethernet and Myrinet networks.

5.2 Gandalf Implementation

The implementation of containers is close to the one presented in [10]. We use four software components : (1) the container interface manager, (2) a container manager, allowing to create and destroy containers, (3) a page manager, using a static distributed manager algorithm and (4) a page server sending or invalidating pages.

Figure 8 presents a functional diagram of containers and memory and file linkers in GOBELINS.

5.2.1 Handling a Page Fault in GOBELINS

When a page fault occurs **[1]**, the LINUX page fault handler is activated and calls the *do_page_fault* function **[2]**. The execution of this function leads to the calls of the memory interface linker *memory_linker_first_touch* function **[3]**. If the requested page is not present on the local node, the container *get_page* function in called **[4]** which sends a request to the page manager **[5]** to get a readable copy. If no copy of this page exists in the cluster and the page belongs to a memory container, a message is sent to the requesting node to create a new page **[6]**. This one then call the associated I/O linker **[7]**, which allocates a page frame with the dedicated LINUX kernel function **[8]**.

In the case of a file container, the page manager sends a request to the node hosting the file **[6b]** to create a copy thanks to the file I/O linker **[9]**. This one requests a page read to the LINUX file system **[10]**.

Lastly, if a page copy exists, the page manager forwards the page request to the owner **[11]**, which sends back a copy of the page to the requesting node.

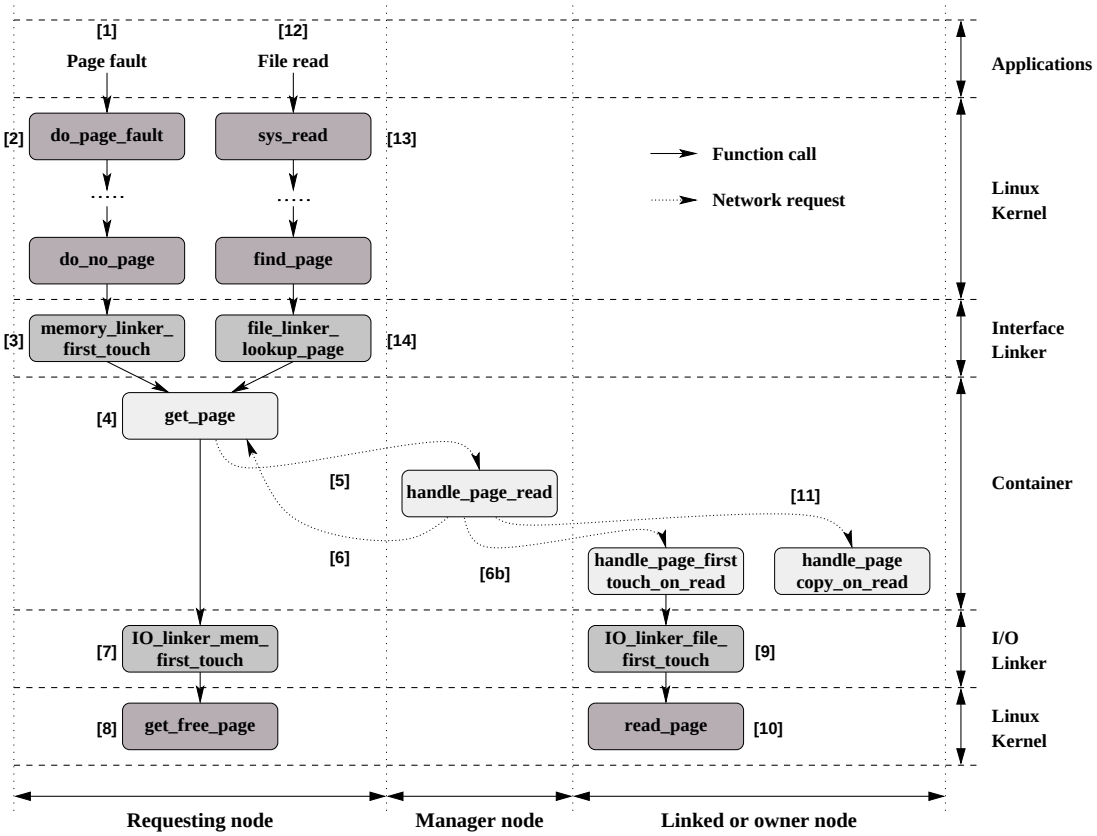


Figure 8: *Resolution of a page fault and file access through containers and linkers*

5.2.2 Handling a File Read in GOBELINS

During a call to the read file function **[12]**, a system call invokes the corresponding LINUX kernel function **[13]**. The execution of this function leads to the call of the file interface linker *file_linker_lookup_page* function **[14]**. If the requested page is not present on the local node, the container *get_page* function is called **[4]**. This function sends a request to the page manager **[5]** to get a readable copy. The following of the read process is then identical to what presented for the resolution of a page fault.

6 Performance Evaluation

We present now, a performance evaluation of the container mechanism implemented in the GOBELINS operating system. The experimentation platform is made up of 6 PCs interconnected by a Gigabit Ethernet network. Each PC consists of a Pentium III 500 MHz processor, 512 MB of main memory and a SCSI IBM Ultra Star 9 GB hard drive.

6.1 Performance of the SVM

First, we have evaluated the performance of containers when they are used to implement a shared virtual memory. We used a set of parallel algorithms programmed using POSIX threads according to a shared memory model.

6.1.1 Evaluation Methodology

Two algorithms were used: *MGS* and *Jacobi*. *MGS* is an algorithm producing a orthonormal basis from an independent set of vectors, using the modified Gram-Schmidt algorithm. For each iteration, a new vector of the basis is computed by a processor and used by all other processors to correct the remaining vectors to standardize. A cyclic distribution of vectors is used.

Jacobi is an algorithm used to solve an Helmholtz differential equations system using the Jacobi iterative method. For each iteration, the matrix calculated during the previous iteration is copied in a temporary matrix. This temporary matrix is then used to calculate the elements of a new matrix. Each element (i, j) of this new matrix is calculated thanks to the element (i, j) of the old matrix and the four neighbor elements. A block cyclic distribution of lines is used.

For each algorithm, we varied the data set size from 64*64 to 2048*2048 double precision floating point elements and varied the number of node from 2 to 6.

6.1.2 Results

Figure 9 shows the speed-up obtained with the Jacobi algorithm. The speed-up increases quickly according to the size of the problem. Moreover, for a given problem size, the speed-up moves away quickly from the optimal speed-up when the number of nodes increases. Nevertheless, from a problem of size 512*512, the speed-up is stable when the problem size or the number of nodes increases.

Figure 10 shows the speed-up obtained with the MGS algorithm. The speed-up increases slowly according to the size of the problem. Moreover, for a given problem

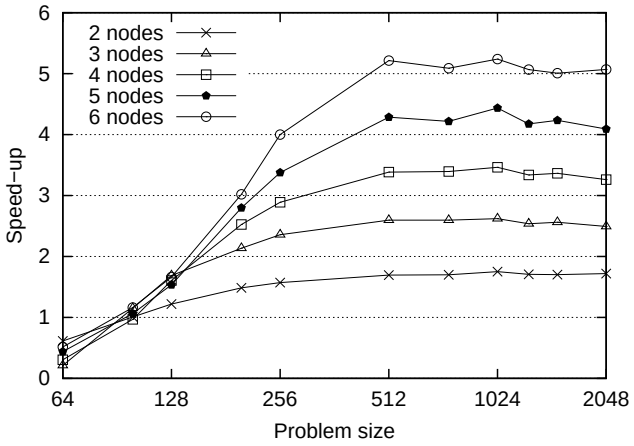


Figure 9: *Speed-up with Jacobi Algorithm*

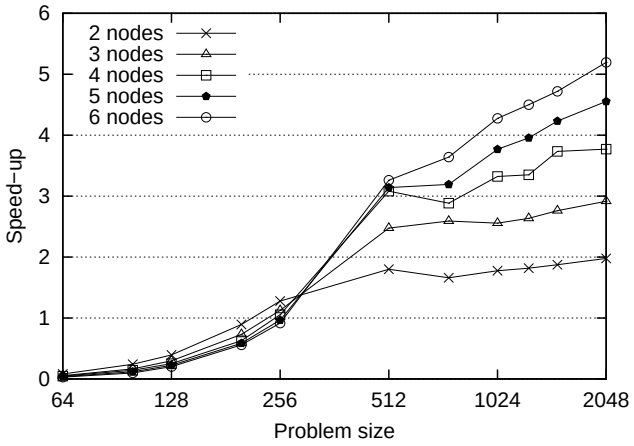


Figure 10: *Speed-up with Modified Gram-Schmidt Algorithm*

size, the speed-up moves away quickly from the optimal speed-up when the number of nodes increases. Nevertheless, the speed-up reaches 5,2 on 6 nodes with a problem size of 2048*2048.

6.2 Performance of the Cooperative File Cache

We have studied the performance of containers when they are used to implement a cooperative file cache. We consider the case of a sequential application running on a cluster node and reading a file.

6.2.1 Evaluation Methodology

We carried out three different evaluations, each one using one of the following conditions:

1. The file is read from the local disk thanks to the standard LINUX file system (Ext2FS).
2. The file is read through NFS from a remove server located outside the cluster.
3. The file is read through the GOBELINS cooperative cache.

For each experiment, the file cache of the execution node is emptied before the execution. Each test has been carried out for a sequential and a random file read.

6.2.2 Results

Figures 11 and 12 shows the bandwidth measured for the three situations described previously in the case of a sequential and a random access. In the case of a sequential read, the peak bandwidth is 14 MB/s for a read from the local disk, 2 MB/s for a read from NFS and 60.9 MB/s for a read from the GOBELINS cooperative cache (assuming the file as already been read on a remote node).

In the case of a random read, the peak bandwidth is 1.8 MB/s for a read from the local disk, 0.5 Mo/s for a read from NFS and 60.9 MB/s for a read from the GOBELINS cooperative cache. It is interesting to see that the cooperative cache is completely access order insensitive. The access latency to a page located in the cooperative cache is constant whatever this page is. In the case of a disk, the loading latency for a page strongly depends on the localization of this page on the disk compared to the current position of the read/write head. Any seek of the read/write head induces a significant increase of the latency. In the case of a random read, the cooperative cache offers a bandwidth more than 30 times higher than the one obtained with a read from the local disk.

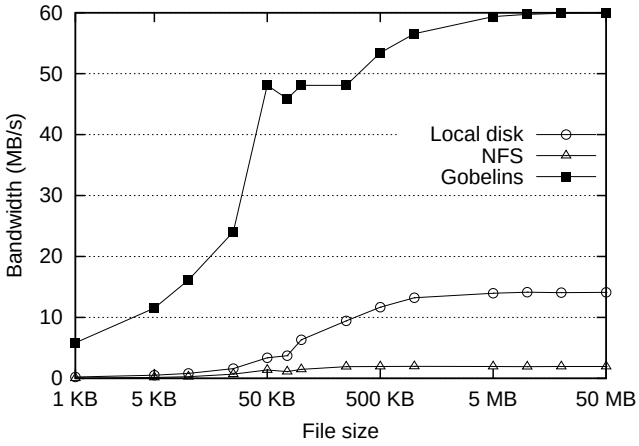


Figure 11: *Bandwidth with a sequential read*

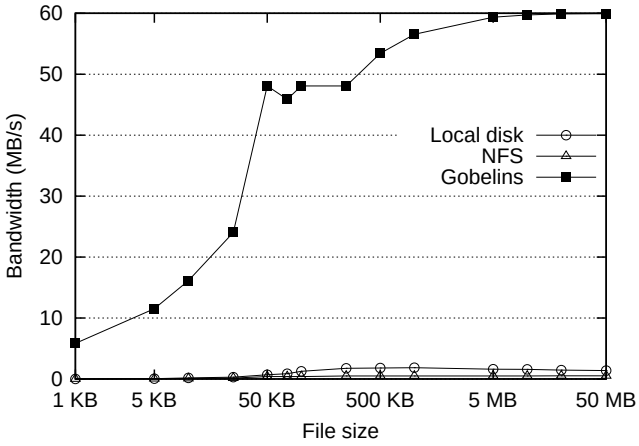


Figure 12: *Bandwidth with a random read*

6.3 Process Migration Performance

In this section, we study the performance of a process migration mechanism based on containers.

6.3.1 Evaluation Methodology

Two different migration mechanisms have been implemented in our prototype: one using containers to ease migration and one using specific mechanisms migrating the whole process address space during process transfer phase. The former is called *Gobelins migration* and the latter is called *basic migration* in the remainder of this paper.

For our evaluation, a sequential application executing the MGS algorithm has been used. We made a set of measurements on various sizes of the matrix used in MGS. Migration is activated at an arbitrary time during the application execution.

6.3.2 Results

Figure 13 shows the migration cost for Gobelins and basic process migration mechanisms. This cost represents the time needed to restart a process on the remote node. We can see that this cost is constant for the Gobelins migration, about 83 ms, whereas the cost of the basic migration increases with the number of memory pages used.

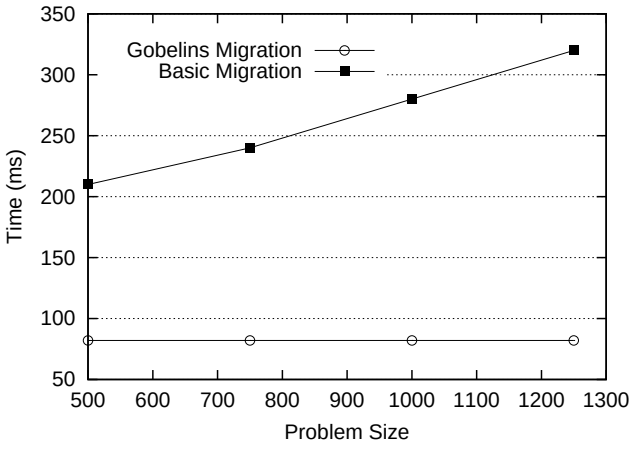


Figure 13: *Migration cost with Gobelins and Basic Migration*

Figure 14 shows the overhead of migration, for Gobelins and basic migration. The overhead is calculated as the difference between the total application execution

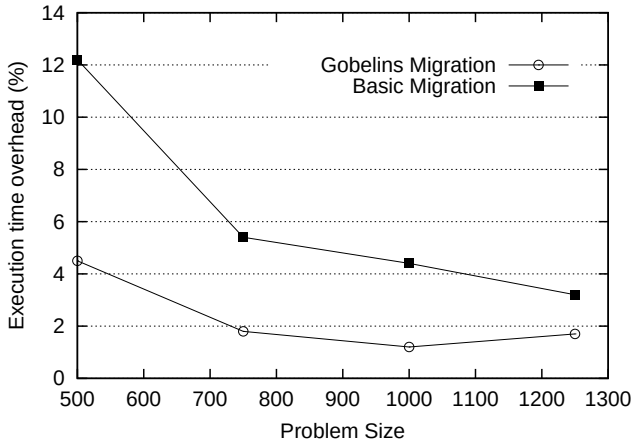


Figure 14: *Migration overhead with Gobelins and Basic Migration*

time obtained without migration and the total execution time obtained when the executing process migrates once.

Results show that Gobelins migration is more efficient than the basic migration mechanism. The overhead due to data transfer through containers is lower than the overhead incurred by the transfer of the whole process address space. With Gobelins migration, pages being migrated on demand, only useful pages are migrated, decreasing the overall amount of data transfered.

7 Conclusion

We have presented in this paper a new architecture for the design of a single system image cluster OS, thanks to concepts of container and linker. The container mechanism combined with a set of linkers makes it possible to integrate several distributed system services within an existing operating system with very few modification of this system. A prototype based on Linux has been built, but the genericity of the container concept makes it possible to consider its use in other operating systems, such as Solaris or Windows NT.

Performance results with GOBELINS show that a shared virtual memory implemented thanks to containers makes it possible to reach good speed-ups for the execution of parallel applications on a cluster. For a cooperative file cache, results obtained

indicate that it is possible to reach a very significant performance improvement compared to the access from the local disk or through a file server such as NFS. This performance improvement is even more significant when the file access is irregular. Finally, performance evaluation of a process migration mechanism using containers shows that migration cost is lower than a migration mechanism using specific full address space migration, still easing the migration design.

Future works includes evaluating performance of the GOBELINS operating system with real large scale industrial applications and to continue on implementing distributed services on top of containers.

References

[1] [http://ssic-linux.sourceforge.net/].

[2] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.A. Patterson, D.S. Roselli, and R.Y. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 285–302. ACM Press, December 1995.

[4] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System*, volume 672 of *Lecture Notes in Computer Science*. Springer, 1993.

[5] M.D. Dahlin, R.Y. Wang, T.E. Anderson, and D.A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating System Design and Implementation*, November 1994.

[6] M. J. Feeley, W. E. Morgan, F. H. Pighin, A. R. Karlin, and H. M. Levy. Implementing global memory management in a workstation cluster. In *Proc. of the 15th ACM Symp. on Operating Systems Principles (SOSP-15)*, pages 201–212, December 1995.

[7] Roy Friedman, Maxim Goldin, Ayal Itzkovitz, and Assaf Schuster. MILLIPEDE: Easy parallel programming in available distributed environments. *Software Practice and Experience*, 27(8):929–965, 1997.

[8] A.M. Goscinski, M.J. Hobbs, and J. Silock. Genesis: The operating system managing parallelism and provinding single system image on clusters. Technical report, TR C00/03, School of Computing and Mathematics, Deakin University, February 2000.

[9] Erik Hagersten, Anders Landin, and Seif Haridi. DDM — A cache-only memory architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[10] K. Li. *Shared Virtual Memory on a Loosely Coupled Multiprocessor*. PhD thesis, 1986.

[11] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computers*, 7(4):321–359, November 1989.

[12] John K. Ousterhout, A. R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *Computer*, 21(2):23–36, February 1988.

[13] Eduardo Pinheiro and Ricardo Bianchini. Nomad: A scalable operating system for clusters of uni and multiprocessors. In *Proceedings of the 1st IEEE International Workshop on Cluster Computing*, December 1999.

[14] R.Lottiaux. *Gestion globale de la mémoire physique d'une grappe pour un système à image unique : mise en œuvre dans le système Gobelins*. PhD thesis, IRISA, Université de Rennes 1, December 2001.

[15] R.Lottiaux and C.Morin. Containers : A sound basis for a true single system image. In *Proceeding of IEEE International Symposium on Cluster Computing and the Grid*, pages 66–73, May 2001.

[16] G. Vallée, C. Morin, J.Y. Berthou, Y. Dutka Malen, and R. Lottiaux. Efficient process migration based on gobelins distributed shared memory. In *To Appear in the Fourth International Workshop on Software Distributed Shared Memory*, may 2002.

[17] Bruce Walker and Douglas Steel. Implementing a full single system image unixware cluster: Middleware vs underware. In *Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'99*, 1999.

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399